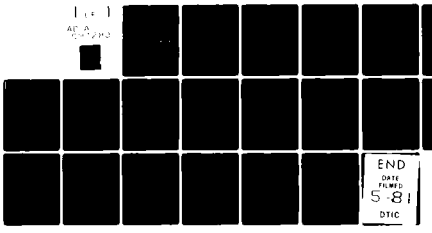
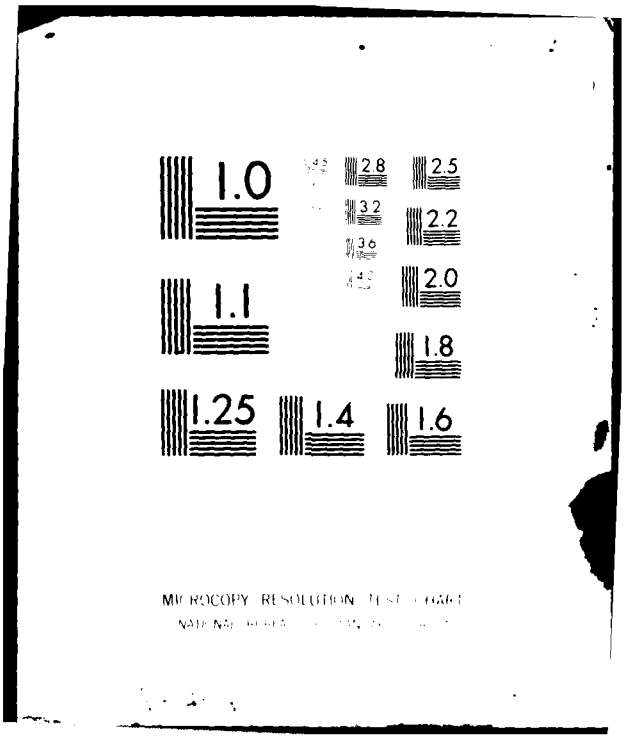
AD-A097 280 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
APPLICATIVE SPECIFICATIONS OF DISTRIBUTED SYSTEMS: EXTENDING TH--ETC(U)
MAR 81 P ZAVE AFOSR-77-3181

UNCLASSIFIED AFOSR-TR-81-0314 NL



END
DATE FILMED
5-81
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LEVEL II

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER (19) AFOSR-TR-81-0314	2. GOVT ACCESSION NO. AD-A097280
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) (6) APPLICATIVE SPECIFICATIONS OF DISTRIBUTED SYSTEMS: EXTENDING THEM TO EMBEDDED SYSTEMS	5. TYPE OF REPORT & PERIOD COVERED: INTERIM
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Pamela Zave (15)	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3181
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, University of Maryland, College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (16) (17) 2304/A2 61102F
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB DC 20332 (11)	12. REPORT DATE: MAR [redacted]81
	13. NUMBER OF PAGES: 29 (12) 32
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): (9) Interim technical rept.	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC ELECTE APR 2 1981 S D B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Recent extensions to applicative languages have made it possible to use them, with their numerous theoretical and practical advantages, for specifying the requirements and designs of distributed systems. Embedded systems are an increasingly important class of distributed systems, and have special problems not adequately dealt with by existing applicative languages. This paper proposes further extensions, shows how they are used to specify embedded systems, and discusses the ramifications of these extensions on the fundamental assumptions of applicative programming.

DD FORM 1 JAN 73 1473

UNCLASSIFIED 409022

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD A097280

DTIC FILE COPY

APPLICATIVE SPECIFICATIONS OF DISTRIBUTED SYSTEMS: EXTENDING THEM TO EMBEDDED SYSTEMS*

Pamela Zave
Department of Computer Science
University of Maryland
College Park, Maryland 20742
U.S.A.

(301) 454-4251

Abstract Recent extensions to applicative languages have made it possible to use them, with their numerous theoretical and practical advantages, for specifying the requirements and designs of distributed systems. Embedded systems are an increasingly important class of distributed systems, and have special problems not adequately dealt with by existing applicative languages. This paper proposes further extensions, shows how they are used to specify embedded systems, and discusses the ramifications of these extensions on the fundamental assumptions of applicative programming.

*This research was supported in part by the Air Force Office of Scientific Research (AFOSR-77-3181B). Computer time was provided by the Computer Science Center, University of Maryland.

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

Applicative Specifications of Distributed Systems: Extending Them to Embedded Systems

0. INTRODUCTION

This paper considers the domain of large, complex, special-purpose computer systems. These systems are often distributed, either because some long-distance communication is a necessary part of their functionality, or because the computational power of multiple processors must be brought to bear on the problem. Early development of these systems should always proceed on the assumption that the ultimate implementation may be distributed, given the numerous hardware options available today.

There is a great need for specification techniques that can be used to help us understand and communicate about these systems. "Applicative programming" has shown a great deal of promise as a specification technique, and this paper explains why it seems so advantageous (Section 1) and how recent extensions to the fundamental concepts have made applicative specifications of distributed systems possible (Section 2).

The main purpose of this paper, however, is to extend applicative languages even further, so that superior specifications of requirements and designs for embedded systems can be written. Embedded systems are defined as a subclass of large, complex, special-purpose systems, and the proposed extensions are motivated (Section 3) and defined (Sections 3 and 4). Section 5 discusses their ramifications on the fundamental assumptions of applicative languages.

1. APPLICATIVE LANGUAGES AS SPECIFICATION LANGUAGES

1.1. What is a specification? A simple view

The most basic concept of a specification is that it is an interface between human intentions and an implementation (Figure 1, after [Liskov & Zilles 75]). A person uses the specification to express what he wants; for this reason it should be both precise and easy to understand. Through the informal human processes of introspection and communication it is decided that

the specification does indeed capture those intentions successfully, often called "validation".

There is a family of programs which correctly implement the specification. Since it is now commonly agreed that both the specification and the program should be formal objects with well-defined semantics, the program can (in principle) be proven correct ("verification"). This imposes on the specification technique the additional requirement that specifications be easily manipulated and transformed.

Another distinction being implied here is that the implementation "runs" and the specification does not. The specification is supposed to say what to do, and the implementation is supposed to say how to do it.

This concept works best in a domain for which there is a clear, coherent, theoretical model of what is to be done, and it is easily distinguished from the "how". Good examples are Boolean algebra as a specification language for combinational logic circuits, and relations as a specification language for databases.

1.2. A richer view, for more complex systems

Systems of the complexity we are interested in require many more layers of organization. Often separate teams will prepare a set of requirements and a design specification, while yet a third team does the actual implementation. If the system is distributed, the design itself may require several specifications, corresponding to such concepts as application-oriented functions, network communication, and single-computer operating systems (e.g. [Smoliar & Scalf 79]).

The model of Figure 1 can be expanded into multiple layers as shown in Figure 2--but our choice of particular intermediate specifications is for illustrative purposes only, and should not be interpreted as offering a theory of distributed system design. The requirements specification must be validated in cooperation with the customer, and serves as the vehicle of communication to (and perhaps the basis of a contractual relationship with) the design team. The program specification, prepared by the design team, is the input to the implementation phase.

The design specifications illustrate three different types of decomposition of complexity. When requirements are elaborated into a design, or a design specification is elaborated into a program specification,

additional decisions are made and additional detail is added. The former specification, which contained less detail than the latter, was an abstraction. Each level of specification (abstraction) provides the "how" for the level above it, and the "what" for the level below ([Ross & Schoman 77]). The aspects of the new specification which represent new constraints on the system must be validated; the aspects of the new specification which are simply more concrete expressions of what is in the old specification must be verified.

Figure 2 assumes that network design, application design, and resource management design are largely independent, and can proceed in parallel with few integration difficulties at the end. This ambitious form of decomposition is called projection, because the entire system description is being projected onto views, each representing only a subset of the properties of the whole. Finally, the program specification is seen as the sum of its parts (each of which can be given to a different programmer), i.e. it is decomposed through partition.

The concept of specification shown in Figure 2 is closer to what we are looking for, but it still has several problems. One is that the administrative and technical goals will often be in conflict: the requirements specification, for instance, should be a clear, coherent, uniformly abstract, and theoretically sound model (from the technical standpoint), but should also include a wide range of legally binding provisos about such things as equipment and maintenance (from the administrative standpoint), as in [Yeh et al. 80]. Common sense indicates that it will be very difficult to achieve both.

Another serious problem is that testing, the most powerful validation tool, cannot be used until the very end--when all validation should have long been completed! Many people have argued that program verification cannot replace testing; what they are saying (in these terms) is that verification from a specification cannot replace validation of the specification. Unfortunately, the specification cannot be completely validated until it has been subjected to testing, which cannot be done until it is translated into executable form.

Our approach to these difficulties is to look for a single specification language that is (a) usable at all levels of abstraction, to express a wide variety of computational views, and (b) executable by an interpreter

regardless of the level of abstraction at which it is used. Such a language would provide maximum leverage in reconciling opposing needs, re-integrating decompositions, and diagnosing problems early by testing the system during all phases of development. One example of the many payoffs is that the requirements specification could be interpreted, thus simulating and displaying proposed system behaviors for the benefit of the customer. His feedback on detailed system behavior could then be solicited before the requirements were considered finished.

In this concept of specification, the question arises: What is the difference between a system and its completed specification, since both run and generate the same behaviors?

The specification language interpreter will probably be a design tool running under a general-purpose operating system on a large timesharing machine. If the speed, reliability, availability, and geographical access offered by this configuration are adequate for the application, then there need be no difference--the system specification (when fully elaborated, so that the only primitives remaining are provided intrinsically) can also be its implementation, via the interpreter. If, on the other hand, the application requires high performance or geographical distribution, the interpreter could never provide a satisfactory implementation, and the system's specification will be clearly distinguishable from its implementation at any level of abstraction.

1.3. Applicative languages

"Applicative" (or "functional") languages are those based on side-effect-free evaluation of expressions formed from constants, formal parameters, functions, and functional operators. Functional operators are "combining forms" for functions, such as composition and conditional selection (the LISP "cond"). Well-known examples of purely applicative languages are the lambda calculus, pure LISP, and the functional programming systems of [Backus 78].

These languages meet all the requirements for specification languages set forth above, with interpretability being the most obvious. A primitive "function" is interpreted as an arbitrary relation from its domain to its range. When it must be evaluated, the interpreter can choose a range value randomly, or perhaps ask a user at a terminal which value should be returned

(thus providing an interactive testing capability).

Applicative languages also have great powers of abstraction, i.e. of decision deferment. The expression "f[(g[y],h[z])]", for instance, tells us that f is to be applied to the values of g[y] and h[z], without constraining the data, control, or processor structures used to do so. Are g[y] and h[z] evaluated sequentially or in parallel? In what temporary data structures are their values stored? Perhaps the arguments y and z are even shipped off to special g- and h-processors, respectively, at different nodes of a network! Furthermore, a primitive function can represent a set of deferred decisions (to be made later by elaborating the function in terms of "smaller" primitives), or a mapping which must remain forever non-deterministic from the viewpoint of the specification (such as the result of an interactive user's thought). Because of these many options, applicative languages have been used successfully to describe phenomena ranging in level of abstraction from distributed system requirements to digital hardware ([Fitzwater & Zave 77], [Smoliar 79]).

Applicative languages are also extremely convenient for formal manipulations such as verification. This is because an expression has "referential transparency", i.e. its only semantic property is its value. An applicative program can be proven consistent with an axiomatic specification of correctness, for example, merely by algebraic substitution! This facility is one of the major subjects of [Backus 78].

It should also be noted that, as programming languages, applicative languages may have more potential for efficient implementation than procedural ones. This is because the "von Neumann bottleneck" of accessing and referring to memory one word at a time has been eluded ([Backus 78]), so that the field is clear for high-powered optimization by interpreter writers and machine designers. The work of Friedman and Wise on large-scale multiprocessing ([Friedman & Wise 78a]) and research on dataflow computers are both efforts in this direction; neither form of parallelism requires the knowledge or participation of the programmer.

1.4. Applicative notations

Applicative languages have a reputation for unreadability in some circles. We will now argue that the properties of applicative notations which cause trouble are optional ones, and that by eschewing them we can present an

applicative language that is usable even for large-scale software engineering.

The most important of these properties is typelessness: most commonly, in the applicative programming literature, the only type of data object is the list or sequence, and all functions are applied to one list and produce one list. Since every function should be prepared to accept argument lists of any internal structure, there must be a distinguished "undefined" value produced whenever the internal structure of the argument is unsuited to the semantics of the function (as in [Backus 78])--and this mismatch must be detected! Multiple arguments to or values from functions must be packaged in single lists, yet the existence of this substructure (or any other substructure, for that matter) cannot be explicitly acknowledged.

Of course, deliberate substructure in data items is ubiquitous, and it is common practice to document it with the use of data types. Furthermore, typing in a language provides a useful form of redundancy which is susceptible to automated checks of internal consistency.

In the applicative notation used in this paper, names of sets (analogous to data types) are strings of capital letters and hyphens ("ONE-SET", "ANOTHER-SET"), and non-primitive sets can be defined using set expressions ("NEW-SET = 'set expression'"). Set expressions use set names and set union ("A U B"), cross-product ("A x B"), and enumeration ("{ true, false }"). Non-numeric constants are underlined, and integer values are available in the intrinsic set INTEGERS.

Function names use small letters and hyphens ("new-function"), and the domain and range of every function, primitive or not, must be declared (although a function need not have arguments). The domain and range declarations can use arbitrary set expressions. Here are three example declarations:

f: ---> A

g: B x C ---> D U E

h: S ---> T.

A function is applied to an argument using the syntax "new-function[x]", and the type of x must be consistent with the domain declaration of new-function. Consistency can be defined with the assistance of type conversion, however, so that the composition "h[g[f]]" is perfectly legal if the definitions:

A = B x C

S = INTEGERS

D = { 0, 2, 4, 6, 8 }

E = { 1, 3, 5, 7, 9 }

have been made. This notion of typing provides all the documentation and redundancy desirable for engineering goals, without sacrificing any of the flexibility attributable to typelessness. All that it requires is the ability to compare any two set expressions for containment, which is easily done given this particular language of set expressions.

A way to gain simplicity in an applicative language is to restrict the functional operators, just as in structured programming the control structures are restricted. In our language non-primitive functions are defined using functional expressions ("f = 'functional expression'"), and functional expressions can contain function applications and compositions, constructions ("(g[y],h[z])"), and conditional selections ("[p1:f1,p2:f2, . . . ,true:fn]", evaluating to the first fi such that the predicate pi is true). Substructures of the argument can be identified using formal parameters ("g[(y,z)] = [p[y]:even[z],true:odd[z]]"), as an alternative to selectors such as "car" and "cdr" in LISP.

Other notations will be introduced as needed. We prefer the style of using special characters for all delimiters, functional operators, and other syntactic structures. The resulting specifications are concise, and very readable because all words appearing anywhere in them are user-chosen names.

Throughout the paper mappings will be called "functions", despite the fact that mappings named in specifications are often relations. The reason is simply that "function" sounds more natural; its use is justified on the grounds that the intention is always to produce a unique value when the mapping is invoked at runtime, even though that value cannot always be defined by a known functional expression.

2. APPLICATIVE SPECIFICATIONS OF DISTRIBUTED SYSTEMS

At first glance it may not seem possible that a language containing nothing but expressions could be capable of specifying whole systems, especially distributed ones, but this is indeed the case. In this section the major developments facilitating such specifications are surveyed.

The notation introduced in 1.4 will be the "base language" of this paper. This section will present a set of extensions to it, as will Sections 3 and 4.

The two sets of extensions should be viewed as independent, since they differ in spirit, but overlap considerably in the structures they make it possible to specify. There is no reason, however, that the two sets could not be used together, as long as careful type-checking were provided to ensure local consistency.

2.1. History-sensitive, cyclic computation

Most complex systems exhibit behavior which is influenced by the history of past events, and many of them are designed to repeat certain cycles of behavior forever. These properties can be specified in a purely applicative expression by adding streams ([Friedman & Wise 78b]), and the functional operators of transitive closure and extension.

A stream is an infinite sequence; if ITEM is a set, then a stream of members of this set will be denoted "ITEM*". Computation on a stream is a perfectly well-defined notion, as long as no attempt is made to use anything but initial subsequences of it. Streams are inputs to perpetual, cyclic computations, and model such phenomena as the stream of characters sent from a terminal to a timesharing system.

Perpetual, cyclic operations on streams are specified using the functional operators of transitive closure and extension. Extension simply applies its function to each member of the stream in turn. Thus "process![s]", where s is of type ITEM, applies the function named "process" to each item, producing as its value a stream of processed items. History-sensitivity is introduced using the transitive closure functional, which repeatedly applies its function to the previous value of that function. Thus "f*[x]" produces the stream (x,f[x],f[f[x]],f[f[f[x]]], . . .).

To illustrate the use of these additions, we will specify a database which is formed by processing a stream of updates. The following are the definitions required:

```
database:
     UPDATE*
---> DATABASE*
database[u] =
     select-database![cyclic-updating[u]]

cyclic-updating:
     UPDATE*
---> (DATABASE x UPDATE*)*
cyclic-updating[u] =
     next-update*[(initial-database,u)]

initial-database:
```

```
---> DATABASE

next-update:
     DATABASE x UPDATE*
---> DATABASE x UPDATE*
next-update[(d,u)] =
     (perform-update[(d,first[u])],rest[u])

perform-update:
     DATABASE x UPDATE
---> DATABASE

select-database:
     DATABASE x UPDATE*
---> DATABASE
select-database[(d,u)] =
     d
```

The "database" function transforms a stream of updates into a stream of databases, each successive database reflecting the incorporation of one more update. The perpetual computation is specified within cyclic-updating, which repeatedly applies next-update to the current database and the stream of updates yet to be processed. Next-update produces a new database value and a new stream of updates yet to be processed. The new database value is obtained by picking the first update off the stream and performing it on the current database. The new stream of updates is simply the old stream with its first update removed ("first" and "rest" are intrinsic functions analogous to car and cdr, respectively).

2.2. Non-determinism based on relative rates

Another important feature of real systems is rate-dependent behavior. It is common throughout operating systems, for instance, that the next action to be taken depends on which of a group of activities in progress is the first to complete.

Such behavior can be modeled in LISP-like languages with the addition of a new data structure called the multiset ([Friedman & Wise 79a], [Smoliar 79]). A multiset is an unordered collection of objects, but differs from an ordinary set because duplicate objects may be present. Just as a list in LISP is formed by applying "cons" to an element and a list, a multiset is formed by applying "frons" to an element and a multiset.

Both lists and multisets are probed with the functions "first" and "rest". The difference is that, while the values of first[l] and rest[l] for any list l are fully determined, the value of first[m] for a multiset m may be any member of m. However, once first[m] has been evaluated, any subsequent evaluation of first[m] will produce the same result, and the value of rest[m]

will be complementary, i.e. contain all members of m except first[m]. In other words, successive probing with first will transform a multiset into a list.

The non-determinism in the multiset concept can be used to model rate-dependence. This is illustrated for a multiprogramming scheduler in [Friedman & Wise 79b]. Multiset operations can also be used to form an update stream for our database as a rate-dependent merge of update streams from three terminals, as follows:

```
rate-dependent-merge:
     UPDATE* x UPDATE* x UPDATE*
---> UPDATE*
rate-dependent-merge[(a,b,c)] =
     sequentialize-by-rate[make-multiset[(a,b,c)]]

make-multiset:
     UPDATE* x UPDATE* x UPDATE*
---> UPDATE* . UPDATE* . UPDATE*
make-multiset[(a,b,c)] =
     frons[(a,frons[(b,frons[(c,null)])])]

sequentialize-by-rate:
     UPDATE* . UPDATE* . UPDATE*
---> UPDATE*
sequentialize-by-rate[m] =
     cons[(first[first[m]]
          'sequentialize-by-rate
               [frons[(rest[first[m]]
                        'rest[m]
                      )
                     ]
               ]
          )
         ].
```

One new item of notation has been introduced here: Just as "A x B" denotes a set of lists (each of whose members has a member of A followed by a member of B), "A . B" denotes a set of multisets, each containing one member of A and one member of B.

2.3. Interaction with the environment

A useful system interacts with its environment, and it may be valuable to have an explicit specification of that environment--especially when implementing the interactions over distance is one of the main complexities of the system. This can be done, however, simply by incorporating the environment into the applicative expression specifying the system.

The environment of our database system, for instance, is the collection of terminals from which its updates come. A terminal can be modeled as a function which provides a stream of updates:

```
terminal:
---> UPDATE*,
```

and defined as the transitive closure of a non-deterministic primitive function, "think-of-update", which represents the thought processes of the person typing in the updates. Thus the rest of the terminal specification is:

```
terminal =
     think-of-update*[null-update]

think-of-update:
     UPDATE
---> UPDATE,
```

and the following expression includes the entire system we have been building, from database to terminals:

```
database
     [rate-dependent-merge
          [(terminal,terminal,terminal)]].
```

Note that it makes sense to have three instances of the "terminal" function, because it is non-deterministic, and will return different update streams in each instance.

This specification is still quite simple, because (among other things) no explicit feedback is modeled. In reality, a system must have an effect on its environment through feedback. In the case of any on-line database system, for instance, input requests are partially determined by the results of previous transactions.

Feedback can also be specified applicatively, using the technique of mutual recursion ([Landin 65]). Without going into the notation, we can see the effect in Figure 3, where the example is text-editing from a terminal. Feedback is introduced by the fact that next-command, mostly representing the user's thought, has as an argument the response to the last command. Next-command can be evaluated as soon as the value of initial-response is ready. The value produced by next-command provides the *enabling argument* for the first evaluation of edit-command, whose value provides the argument for the second evaluation of next-command, etc.

2.4. Distributed computation

The same property that makes applicative languages good for specification in general makes them good for distributed computing: they do not constrain evaluation structures, and this extends to where evaluation resources are, and how arguments and values are moved among them. One dramatic example of the results of this freedom is suspended evaluation and its concomitant

opportunities for large-scale multiprocessing ([Friedman & Wise 78a]). Another is the use of the "apply to all" functional, which specifies that the same function is to be applied to all members of a set or list, to represent parallel processing with dynamic resource allocation ([Smoliar 79]). Doubtless many others will appear as interest in applicative languages continues to grow.

3. EMBEDDED SYSTEMS

Common examples of embedded systems are industrial process control systems, flight guidance systems, communication systems, defense systems, and patient-monitoring systems. In this section we discuss the special characteristics of embedded systems, and the unmet demands they make on our applicative specification language.

3.1. Characteristics and problems of embedded systems

The term "embedded" was coined by the U.S. Department of Defense in conjunction with its common programming language effort. The DoD's experience with embedded systems has been that they are indeed large, complex, potentially distributed, and require the utmost in software engineering effort ([Fisher 78]).

"Embedded" refers to the fact that these systems are embedded in larger systems whose primary purposes are not computation, but this is actually true of any useful system. The real difference between an embedded system and a data-processing system is that an embedded system interacts with its immediate environment, which is usually inanimate, in such a way that non-trivial, relatively inflexible, performance constraints are required (this is discussed at length in [Zave 80b]). Thus "embedded" is almost synonymous with "real-time", but we prefer the newer term because it does not exclude performance requirements dealing with reliability.

Another important characteristic is that the interface between an embedded system and its environment tends to be complex, asynchronous, distributed, and often not reproducible for testing purposes (the environments of ballistic missile defense systems or spaceflight software, for instance). This can make it especially difficult to specify precise requirements for an embedded system, or to determine if a given implemented system meets them.

3.2. Modeling the environment

Explicit modeling of the environment, as we modeled the terminals in the environment of the database system, can do a great deal to alleviate specification problems associated with embedded systems. Explicit interaction between an environment model and a proposed system model is a much clearer way to specify a complex interface than to treat one side of it as a "black box". The environment model is also the appropriate place to attach many performance requirements (the load on an on-line database system, for instance, is dependent only on the number and output rate of the terminals attached). If the environment model is executable (as a model in an applicative language would be), it can serve as a simulated test driver for a developing system which perhaps cannot be tested in any other way. Furthermore, there is good reason to believe that an environment model will be beneficial to the thought processes of requirements analysts ([Yeh & Zave 80]).

The problem is that the environment/system interface for many embedded systems cannot be specified in current applicative languages. Airplanes, spacecraft, machines, and sick people all share the property of being "free-running", i.e. they continue in their courses of behavior quite independently of any attempts by computer systems to keep up with them. They will continue to move, be sick, etc., and the way this happens can be affected by a computer system, but only if the system manages to produce its relevant signals on time.

When system and environment objects are coupled applicatively, however, it is in the nature of that coupling to enforce global coordination of supply and demand with respect to values. In the specification of the database with its terminals, for instance, there is no way that the specification could be altered or re-interpreted to cover the possibility that the terminals produce updates too fast for the database to accept them all. Yet analogous situations are common--and acceptable--in many real-time systems.

3.3. Performance constraints

It is obvious from our identification of the term "embedded" with performance requirements that specifications of embedded systems must have performance constraints attached. In this case the problem with applicative specification is that the only absolute "paths" of processing to which

performance constraints such as time can be attached are the "paths" between a desired value and the startup of the system! For perpetual, cyclic computations, the elapsed time since the system was initiated is neither relevant nor convenient.

The basis for this conclusion is best seen in the work of Friedman and Wise on suspended evaluation in LISP (this is mentioned in all their writings, but the discussion in [Friedman & Wise 77] is "earthiest", and hence most related to performance).

In LISP all data structures are created by cons. When a value is to be "consed" onto a list, however, it need not have been computed--instead, the value can be represented by a suspension, containing pointers to the expression defining the value, and to the environment in which that expression is to be evaluated. A list structure with suspensions is said to be promised. It is made manifest by actually doing the promised computations, and thereby replacing suspensions with values.

One of the main points of [Friedman & Wise 77] is that user files can be created in "promised" form, and turned into "manifest" ones only when the values are actually needed, such as when they are to be printed. This is good news for an operating system designer, because it gives him the freedom to delay much of the computation ordered by an interactive user, and hence to allocate his resources for the maximum satisfaction of all users. Note that the suspended evaluation strategy has nothing whatsoever to do with what is being specified by the user's LISP; it is merely one of the many possible implementation strategies allowed by the LISP semantics.

Back in the realm of embedded systems, however, where performance requirements are absolute rather than relative, each value is specified as a purely applicative expression over the initial arguments to the system. Since computation may be promised rather than manifest, if we want to constrain when a particular value will be ready for use, we can only constrain how long it takes to compute it from the system's initial arguments (which are the only thing we can count on to be in fully evaluated form).

3.4. - Processes

The reason for the difficulties encountered in 3.3 is that applicative specifications have no notion of state. States would be "footholds" for performance requirements: since a state should be a manifest data structure,

performance requirements would only have to refer to the computation since the last system state. The exception to the statelessness of applicative languages, of course, is that there is an initial state containing the values of the initiating arguments.

Needless to say, there are excellent reasons for avoiding states in descriptions of distributed systems. Global states are meaningless at worst, and useless (complex, impossible to determine) at best. But we can effect a satisfactory compromise by introducing local states, which pose no theoretical or practical problems, yet provide enough attachment points to permit adequate formal specification of performance requirements ([Zave 79], [Zave 80a]).

When local states are introduced, an applicative specification is partitioned into a set of processes. Each process has a state space, or set of all possible states, and a successor function applicatively defined on the state space. The process is a representation for perpetual, cyclic local computation; it goes through an infinite sequence of well-defined (and manifest) states, each state being computed by applying the successor function to its predecessor (Figure 4).

Processes have long been used as abstractions of concurrent activities within multiprogramming systems ([Horning & Randell 73]). More recently, processes have been mentioned as a means to introduce history-sensitivity into applicative programming systems ([Backus 78]), and used to specify autonomous natural or digital objects in the environments of digital systems ([Zave 79], [Zave 80a], [Zave 80b]).

We will now convert the database-and-terminal system into processes (Figure 5). The database is encapsulated by a process, and each terminal is specified as a process. Since the succession of process steps is the analogue of transitive closure in purely applicative notation, the terminal is converted into a process simply by declaring that UPDATE is its state space, and "think-of-update" its successor function.

For the database process, DATABASE is its state space, "initial-database" produces its initial state, and its successor function is:

```
database-cycle:
     DATABASE
---> DATABASE
database-cycle[d] =
     perform-update[(d ,receive-update)]

receive-update:
---> UPDATE.
```

This specification has several advantages over the corresponding one in

Section 2. It is considerably simpler because the system is partitioned in time by the states, and in space by the process boundaries.

Another advantage is that it is a more faithful representation of reality. By not constraining process/processor structures, purely applicative notation preserves the maximum freedom for the designer of the proposed system. But the environment of the system has already been "designed" in a certain way--in this case, a terminal is a remote, autonomous device, best specified as a process, and certainly not implementable by any kind of resource-sharing among terminals. It must be construed as a weakness, in this context, that purely applicative notation cannot express this fact.

Recall that the base notation of 1.4, which was designed for use with processes (rather than the purely applicative extensions in Section 2) did not include recursion or unbounded iteration ("_while_ . . . _do_ . . ."), nor could any data structure be of unbounded length. The purpose of this restriction can now be made clear. It guarantees _a priori_ that any computation specified _within_ a process step can be completed in a bounded amount of time, using a bounded amount of resources. This, coupled with the fact that many performance requirements can be specified by constraints within process successor functions ([Zave 79], [Zave 80a]) paves the way to designing embedded systems that are guaranteed to meet their performance requirements. At the same time, the addition of processes to the notation, and the characteristics of embedded systems, prevent these restrictions from being a burden to the specifier.

Needless to say, something very important is still missing: How do processes communicate? There must be an asynchronous interaction mechanism that is (a) general, (b) completely compatible with the applicative framework, and (c) capable of specifying real-time interfaces. Such a mechanism is the subject of the next section.

4. EXCHANGE FUNCTIONS

4.1. Exchanging

Asynchronous interactions between processes can be specified within an applicative framework using three primitive functions, collectively called "exchange" functions.

Places in an applicative expression (defining the successor function of a

process) where interaction is needed can be called "interaction sites". They are specified as functions with user-chosen mnemonic names, domain sets that include all possible output values at that "site", and range sets that include all possible input values.

Thus the function "receive-update" in the successor function of the database process is actually an interaction site. There is no output here (and hence no domain), but the input will be a member of UPDATE.

In specifying a system, it is very convenient to be able to leave interaction sites as primitives for awhile, deferring complete specification of the actual communication. This can be done simply because a primitive function is a perfect abstraction of our interaction facility, which provides for mutually synchronized, two-way communication between two interaction sites.

This is what will happen when a fully specified interaction takes place: two "ready" interaction sites (meaning that their arguments have been evaluated) will "match" (how they match is yet to be explained), and arguments will be exchanged. Each site will then return as its (input) value the argument (output) of the other.

4.2. Matching

Each interaction site must eventually be defined as a primitive exchange function. Since two attributes of the exchange function must be specified, namely a type ("xq", "xr", or "x") and a channel (a user-chosen identifier), we use a hyphenated syntax:

```
receive-update =
     xq-up[null].
```

The string before the first hyphen is the type, and the string after the first hyphen is the channel. A primitive exchange function must always have one argument and one value; we use null, the only member of the intrinsic set NULL, by convention if there is no significant output or input, respectively. (The interaction site function, which is defined in terms of an exchange function, must have a value but need not have an argument, as is the case with *receive-update*.)

Only exchange functions with the same channel can match with each other. "Normal" matches (as opposed to "real-time" matches, see 4.3) take place between an "xq" function and an "x" function, both with the same channel.

There can only be one xq with a given channel (but many x's), so any many-to-one competition situations are specified with x's for the many and an xq for the one. Simply put, if an xq and more than one x, all with the same channel, are ready to interact, then the xq matches non-deterministically with any one of the x's--with the proviso that there must be no lockout, i.e. situations where an x waits indefinitely to be matched while other, more recently initiated, x's keep going before it.

We can now specify fully the interaction between the database process and its terminals. The database has already been elaborated down to its xq. The terminals will use x's, of course, because they must compete for the attention of the database. The successor function of a terminal process must be redefined as:

```
terminal-cycle:
     UPDATE
---> UPDATE
terminal-cycle[u] =
     dispose-of-update[think-of-update[u]]

dispose-of-update:
     UPDATE
---> UPDATE
dispose-of-update[u] =
     proj-2-1[(u,send-update[u])]

send-update:
     UPDATE
---> NULL
send-update[u] =
     x-up[null].
```

"Proj-2-1" is an intrinsic function which projects a pair onto its first element.

Even though xq's and x's are asymmetric with respect to matches, they are symmetric with respect to synchronization--each may have to wait for the other. This is illustrated by the timing diagram of Figure 6.

Figure 7 shows a sample implementation which works well in a distributed environment and automatically prevents lockout. When an x is initiated, a message carrying its argument is sent to the node where the matching xq resides. These messages are queued up (the source of the "q" in "xq") in arrival order. When the xq is initiated, if the queue is empty, it waits until it is not. When the queue is not empty, it removes the first entry as the "match", takes the value stored there as its own value, sends a termination message containing its argument to the matching x, and continues. Computation can continue at the x as soon as the termination message (with its value) is received.

4.3. Real-time interactions

Real-time interactions are specified as exchanges between xr's and x's. The situation is very similar to xq/x matching: there is only one xr (and no xq) with a given channel, there may be many x's to compete for it, and a distributed implementation such as that in Figure 7 will successfully carry out the matching.

The only difference is that if evaluation of an xr is initiated, and there is no pending x to match with it, the xr evaluation does not wait. Instead of exchanging, it returns immediately with its own argument as its value (thus it is always possible to determine if an xr exchanged, by keeping its domain disjoint from the domains of any x's that might match with it). The only modification to the proposed implementation needed for real-time matches is that if the unique exchange function is an xr, and it is initiated when its queue of possible matches is empty, then it does not go into the wait state.

Perhaps the most characteristic of all free-running objects is a real-time clock. It can be specified with the following successor function:

```
clock-cycle:
     TIME
---> TIME
clock-cycle[t] =
     proj-2-1[(increment[t],offer-time[t])]

increment:
     TIME
---> TIME

offer-time:
     TIME
---> NULL
offer-time[t] =
     xr-time[t].
```

To complete the specification, we should add the constraint that each evaluation of clock-cycle takes exactly one unit of time. Satisfaction of the constraint cannot be compromised by synchronization delays, because the only exchange function used is xr. To read the current time, other processes evaluate x-time[null]. Note that with this particular clock, any time value can be read by at most one process.

Here is a functional specification of a digital simulation of a patient, a free-running object in the environment of a patient-monitoring system:

```
patient-cycle:
     PATIENT-STATE
---> PATIENT-STATE
patient-cycle[p] =
     proj-2-1[(simulate-patient[(p,treatment-if-any)],
```

```
                    offer-sensor-data[sensors[p]]
                )]

    treatment-if-any:
    ---> TREATMENT U NULL
    treatment-if-any =
         xr-nurse[null]

    simulate-patient:
         PATIENT-STATE x (TREATMENT U NULL)
    ---> PATIENT-STATE

    sensors:
         PATIENT-STATE
    ---> SENSOR-DATA

    offer-sensor-data:
         SENSOR-DATA
    ---> NULL
    offer-sensor-data[s] =
         xr-data[s].
```

The offering of sensor data to monitoring equipment that may or may not be ready to accept it is similar to the clock's offer of time, but here we have feedback through medical treatment as well. If treatment is being given during a cycle of this process, that evaluation of the interaction site "treatment-if-any" will return a treatment code as input to the simulation. Otherwise the null value given as argument to xr-nurse will be returned.

4.4. Generality, consistency, and comparisons

The generality of exchange functions has been established in numerous substantial examples, including a standard message-passing mechanism, an airline reservation system, a patient-monitoring system, a process control system, and an adaptive, distributed numerical system that has been implemented directly from its functional specification ([Zave & Rheinboldt 79]). It seems to be sufficient for all bounded forms of communication (i.e. not unbounded broadcast).* Since the design goal has been to find a minimal and easily interpreted set of primitives, "higher-level" constructions might have to be written as macros, of course.

Although the two-way synchronization of exchange functions is also found in other recent process interaction mechanisms, such as the Ada rendezvous ([Ichbiah et al. 79]) and Hoare's input/output primitives ([Hoare 78]), it is still a frequent source of questions. Why can't a process, that wants to

---

*It should be noted that the formulation presented here is a simplified version of the original exchange functions ([Fitzwater & Zave 77]). It may yet prove necessary to re-introduce some of the former generality, at the cost of a somewhat more complex implementation.

communicate with another just to give work to it, simply send the work and continue? Why must it wait until the server is ready to do the work?

The answer goes back to our concern with embedded systems and performance, and can be illustrated with the database-and-terminals system. Suppose the xq/x match used to transmit the updates were altered so that (assuming the implementation in Figure 7) evaluation of an x just terminated after sending the initiation message, without waiting for a termination message. Then the speed of the terminals could increase (unchecked by the ability of the system to handle the work), the queue at the database could grow to unbounded lengths, and no bounds on the performance of the system could ever be established.

At the same time, there is nothing wrong with buffering. Any bounded degree of buffering can be specified by introducing a buffer process between the terminals and the database.

Given that synchronization is going to be two-way, it costs very little in the implementation to preserve the possibility of two-way data transfer, although it is seldom used. It also keeps the number of primitives down by a factor of two, since otherwise each of the three exchange functions would have to come in a "sending data" and a "receiving data" version.

Since exchange functions have many similarities to Hoare's input/output primitives, it is instructive to compare them. In Hoare's language, a pair of statements, "P?input" in process Q and "Q!output" in process P, will come together in the same mutually synchronized manner that two matching exchanges do. "Output" is an expression whose value is assigned to the variable "input", assuming appropriate type correspondences. In addition to the relatively unimportant data asymmetry, Hoare's primitives seem to be different from exchange functions in three fundamental ways: (a) There is no way to specify real-time interactions. (b) There is no straightforward way to specify resource sharing, since all "matches" are one-to-one by process name. In Hoare's language a process representing a shared resource must have a separate command for each process with which it can communicate, and guard that command ([Dijkstra 75]) with an input command naming the appropriate process of the many. The guard (and statement) to be executed are chosen non-deterministically, based in part on which of the many processes are ready to communicate. These multiple statements seem distinctly clumsy compared to an xq/x exchange match. Furthermore, the full knowledge each process must

have about the names of the processes with which it communicates makes modularity difficult to achieve. (c) Hoare's primitives belong in a procedural, rather than applicative, framework. The destination of a data transfer, for instance, is specified by an address.

Establishing the internal consistency of a specification with exchange functions requires some attention. The range of an interaction site function must agree with the domains of all those with which it can exchange. Furthermore, precedence constraints caused by nested evaluation structures can cause exchange deadlocks. But the channel of an exchange function has been made a constant attribute rather than an argument to it just so that exchange patterns would yield to static analysis, and simple arguments do establish deadlock-freedom in many common cases.

5. RAMIFICATIONS

5.1. Side-effects

Exchange functions are not purely aplicative: they do not have referential transparency, because process interaction is done by side-effect.

It would not be correct, however, to say that something has been lost. Purely applicative specifications are valuable because they describe functional properties precisely and executably without requiring any decisions about performance and resources. But specifications of requirements for embedded systems must have performance properties, and specifications of designs for any type of system must deal with resources, and we claim that the facilities presented here do so in the cleanest, least painful way.

Furthermore, there is reason to believe that any specification which includes a system's environment loses referential transparency. Consider the text-editing example in Figure 3. Next-command would be evaluated by displaying its argument (the response) on a CRT, and prompting the user to type in the next command. While purely applicative in form, this has an inevitable side-effect on the environment--the user now thinks he has typed in that command, and will not knowingly repeat it. Purely applicative expressions, on the other hand, can be evaluated any number of times without any effect on the functional properties of the system (and will always return the same value).

The existence of side-effects has direct consequences on the evaluation

of subexpressions with exchange functions. They cannot be optimized to avoid evaluation of expressions whose values are not needed, as in the common expression "proj-2-1[(a,b)]", where b is an interaction site.

There is also a potential problem with distributing values obtained by interaction, but the formal parameter mechanism does this nicely. Suppose the effect of

```
[equal[(xq-denom[null],0)]:  divide-check,
 true:                       divide[(numerator,xq-denom[null])]]
```

is wanted, where both usages of the value returned by the exchange are supposed to result from a single evaluation. This can be specified unambiguously by defining "quotient" as

```
quotient[(n,d)] =
     [equal[(d,0)]:  divide-check,
      true:          divide[(n,d)]],
```

and then using it in the invocation "quotient[(numerator,xq-denom[null])]".

5.2. Axiomatic semantics of distributed systems

For sequential programs, algebraic axioms are now used to specify semantic properties. Programs can be verified against these axioms (but much more easily, if they are written in an applicative language). Different programs could also be proven semantically equivalent to one another on this basis, the significant difference between two semantically equivalent programs being in their performance and resource usage.

It now appears possible to do the same thing for parallel and distributed systems. Purely applicative expressions can be used to describe the purely functional (performance-independent, resource-independent, and referentially transparent) aspects of the system, and "programs" which use processes and exchange functions to specify design decisions can be verified against them. The theoretical basis required will be an axiomatization of process semantics in terms of purely functional operators and streams, beginning with the axiomatization of process state succession as an "implementation" of transitive closure.

The implications are tantalizing, for semantically equivalent programs would represent different designs (sets of performance/resource decisions) for the same function. It might then be possible to gain a useful theoretical understanding of design trade-offs, and the transformations by which a requirements specification is turned into a detailed design.

Acknowledgments This paper could not have been written without D.R. Fitzwater, who first thought of exchange functions, or Steve Smoliar, whose correspondence stimulated most of the thinking about applicative programming that went into it.

References

[Backus 78]
John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Comm. ACM 21, August 1978, pp. 613-641.

[Dijkstra 75]
E.W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", Comm. ACM 18, August 1975, pp. 453-457.

[Fisher 78]
David A. Fisher, "DoD's Common Programming Language Effort", Computer 11, March 1978, pp. 24-33.

[Fitzwater & Zave 77]
D.R. Fitzwater and Pamela Zave, "The Use of Formal Asynchronous Process Specifications in a System Development Process", Proc. Texas Conf. on Computing Systems, Austin, Texas, November 1977, pp. 2B-21 - 2B-30.

[Friedman & Wise 77]
Daniel P. Friedman and David S. Wise, "Aspects of Applicative Programming for File Systems", Proc. ACM Conf. on Language Design for Reliable Software, Raleigh, N. Car., March 1977, pp. 41-55.

[Friedman & Wise 78a]
Daniel P. Friedman and David S. Wise, "Aspects of Applicative Programming for Parallel Processing", IEEE Trans. Computers C-27, April 1978, pp. 289-296.

[Friedman & Wise 78b]
Daniel P. Friedman and David S. Wise, "Unbounded Computational Structures", Software--Practice and Experience 8, July-August 1978, pp. 407-416.

[Friedman & Wise 79a]
Daniel P. Friedman and David S. Wise, "Applicative Multiprogramming", Computer Science Tech. Rep. 72, Indiana University, Bloomington, Ind., April 1979.

[Friedman & Wise 79b]
Daniel P. Friedman and David S. Wise, "An Approach to Fair Applicative Multiprogramming", Semantics of Concurrent Computation (G. Kahn, ed.), Lecture Notes in Computer Science 70, Springer-Verlag, Berlin, 1979, pp. 203-226.

[Hoare 78]
C.A.R. Hoare, "Communicating Sequential Processes", Comm. ACM 21, August 1978, pp. 666-677.

[Horning & Randell 73]
J.J. Horning and B. Randell, "Process Structuring", Comp. Surveys 5, March 1973, pp. 5-30.

[Ichbiah et al. 79]
J.D. Ichbiah et al., "Rationale for the Design of the Ada Programming Language", SIGPLAN Notices 14, June 1979, Part B.

[Landin 65]
P.J. Landin, "A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I", Comm. ACM 8, February 1965, pp. 89-101.

[Liskov & Zilles 75]
Barbara H. Liskov and Stephen Zilles. "Specification Techniques for Data Abstractions", IEEE Trans. Software Engr. SE-1, March 1975, pp. 7-19.

[Ross & Schoman 77]
Douglas T. Ross and Kenneth E. Schoman, "Structured Analysis for Requirements Definition", IEEE Trans. Software Engr. SE-3, January 1977, pp. 6-15.

[Smoliar 79]
Stephen W. Smoliar, "Using Applicative Techniques to Design Distributed Systems", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 150-161.

[Smoliar & Scalf 79]
Stephen W. Smoliar and Joe E. Scalf, "A Framework for Distributed Data Processing Requirements", Proc. COMPSAC, Chicago, Ill., November 1979, pp. 535-541.

[Yeh & Zave 80]
Raymond T. Yeh and Pamela Zave, "Specifying Software Requirements", Proc. IEEE 68, September 1980, to appear.

[Yeh et al. 80]
Raymond T. Yeh et al., "Software Requirements: A Report on the State of the Art", Software Engineering, C.V. Ramamoorthy and Charles R. Vick, eds., to appear.

[Zave 79]
Pamela Zave, "Formal Specification of Complete and Consistent Performance Requirements", Proc. Texas Conf. on Computing Systems, Dallas, Texas, November 1979, pp. 4B-18 - 4B-25.

[Zave 80a]
Pamela Zave, "The Operational Approach to Requirements Specification for Embedded Systems", Computer Science Tech. Rep., University of Maryland, College Park, Md., in preparation.

[Zave 80b]
Pamela Zave, "'Real-World' Properties in the Requirements for Embedded Systems", Proc. Annual Washington, D.C. ACM Technical Symposium, Gaithersburg, Md., June 1980, pp. 21-26.

[Zave & Rheinboldt 79]
Pamela Zave and Werner C. Rheinboldt, "The Design of an Adaptive, Parallel Finite-Element System", Trans. Math. Software 5, March 1979, pp. 1-17.

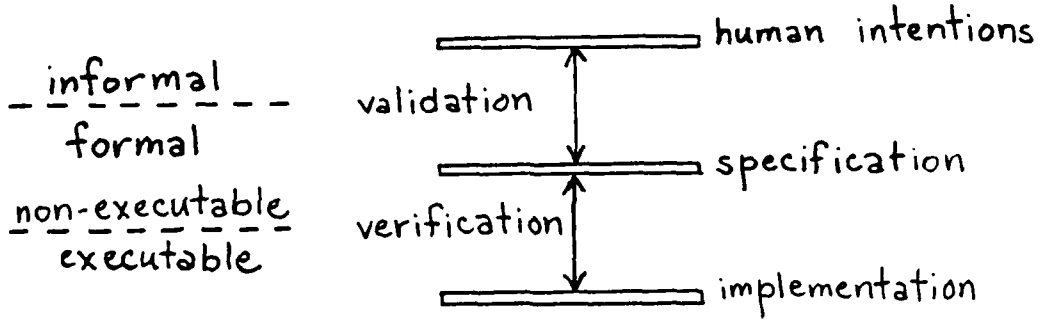


Figure 1. The basic concept of a specification.

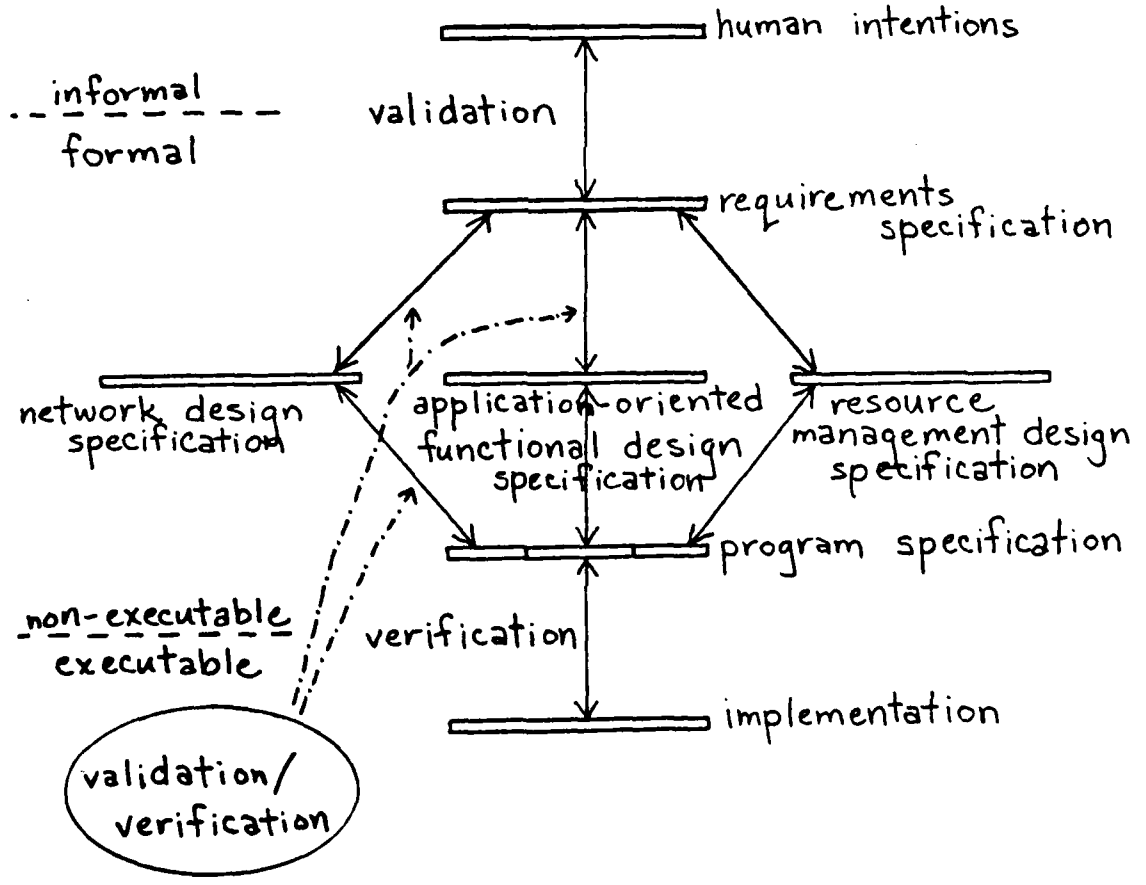


Figure 2. The basic concept of a specification, with multiple layers.

```
edit-command:  COMMAND x TEXT ---> RESPONSE x TEXT
next-command:  RESPONSE ---> COMMAND
```

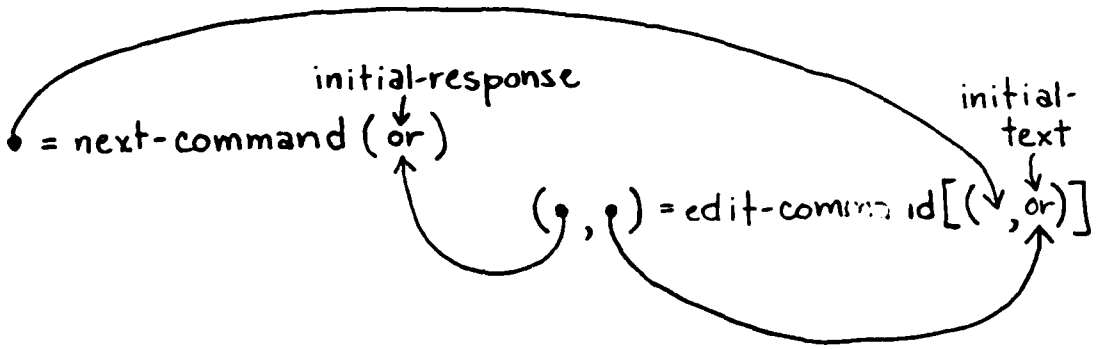


Figure 3. Data flow in a mutually recursive specification of a feedback loop.

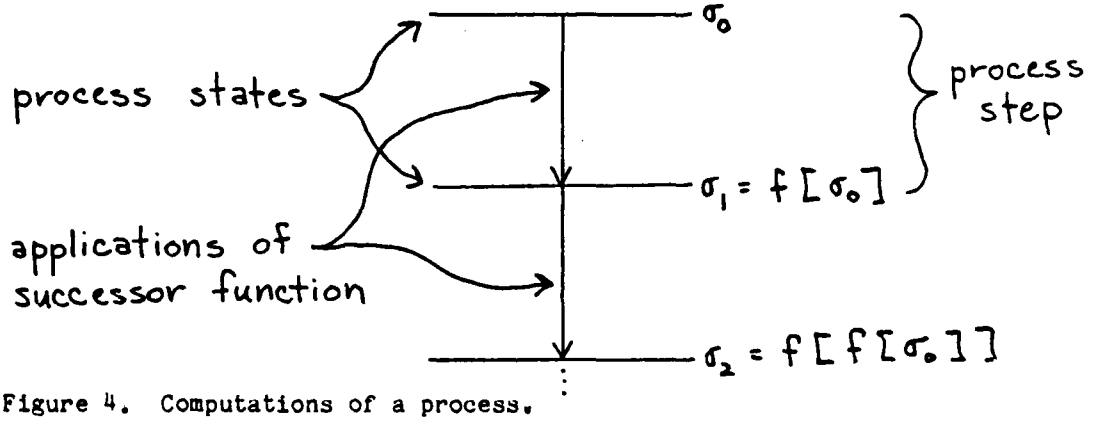


Figure 4. Computations of a process.

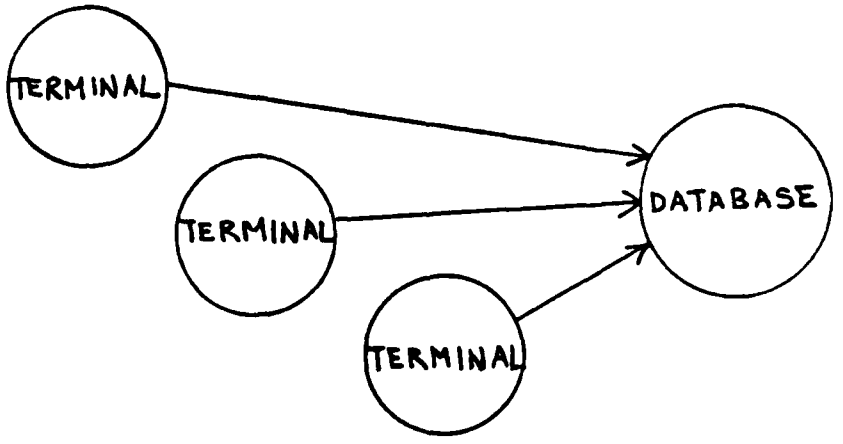


Figure 5. The database system, partitioned into processes.

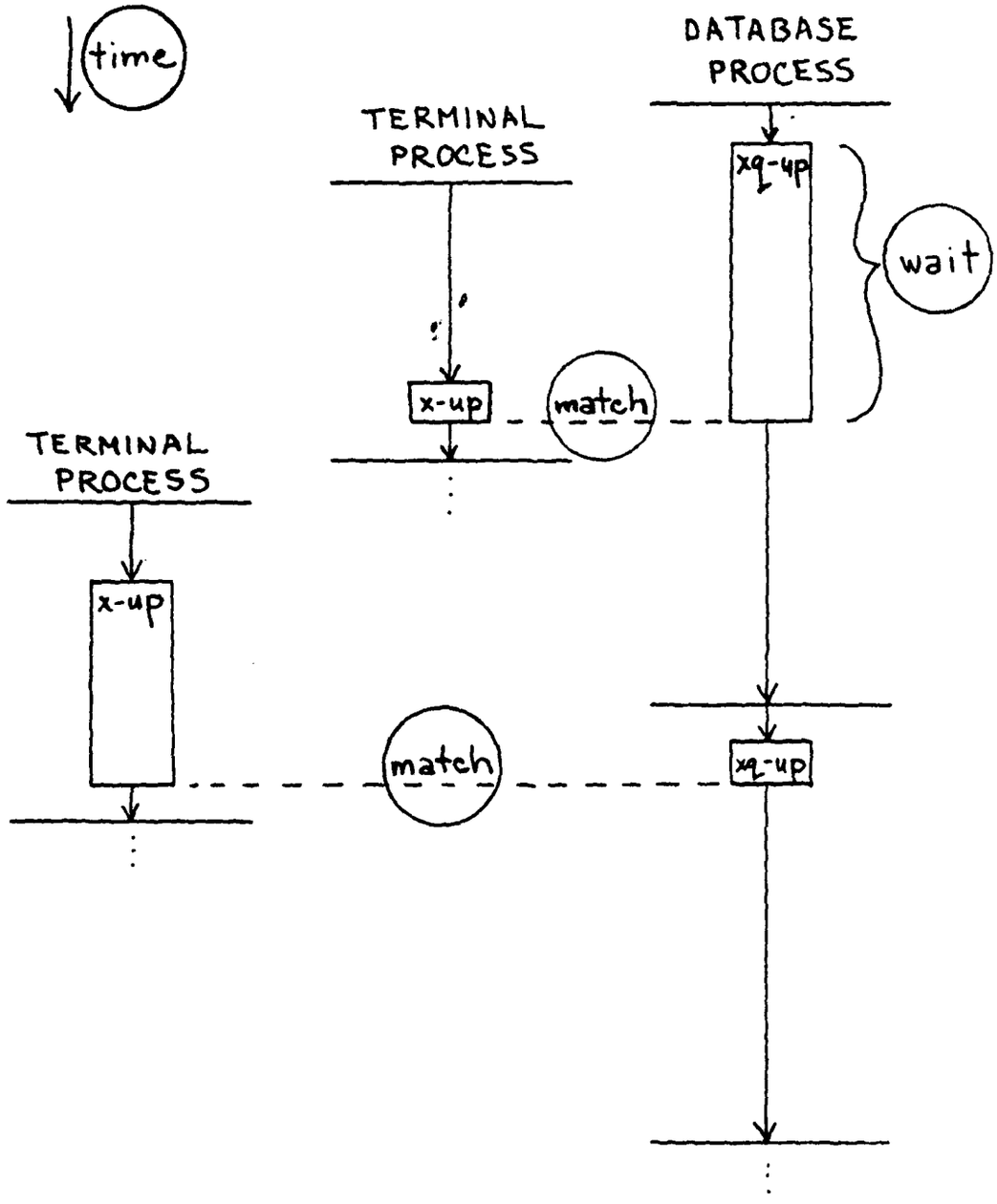


Figure 6. Timing of some database/terminal interactions.

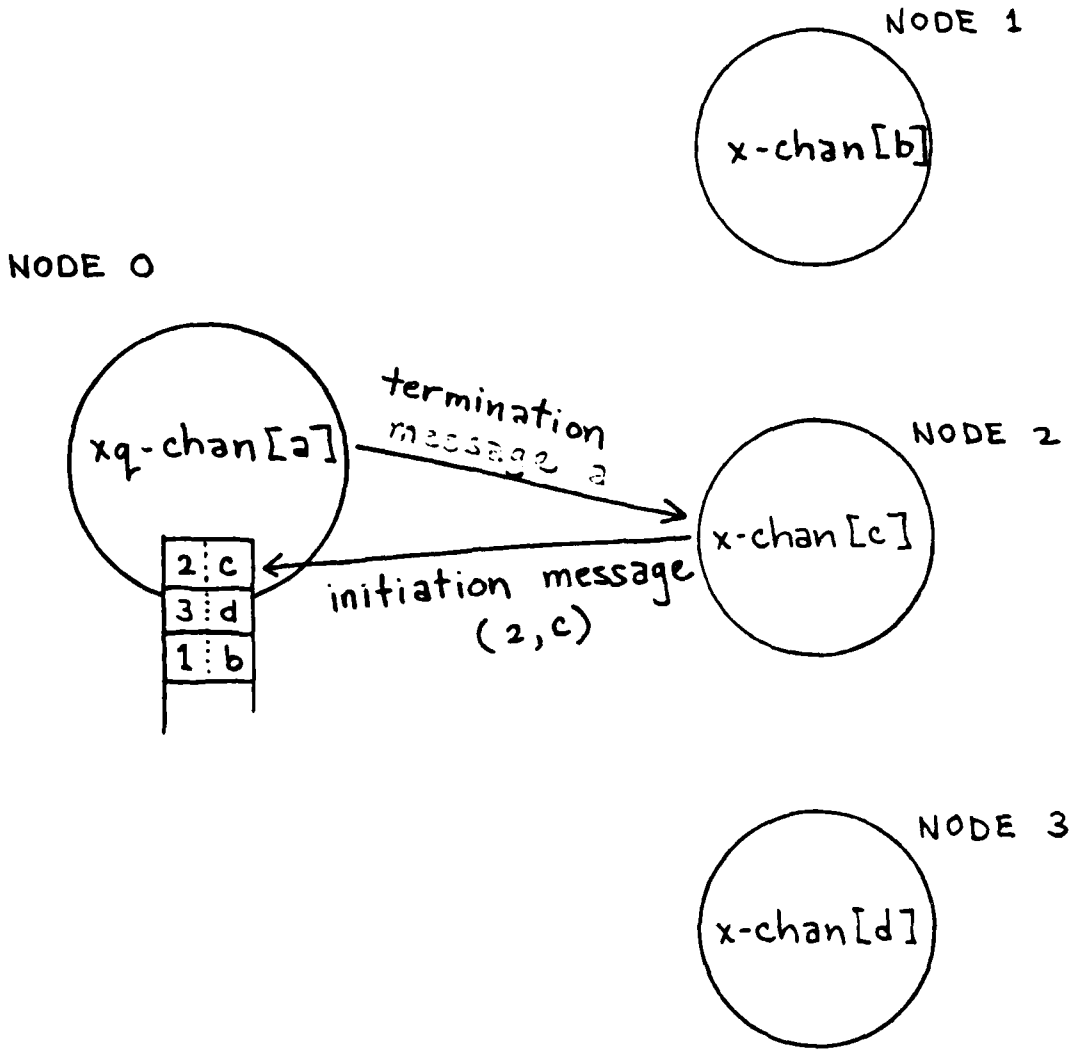


Figure 7. Distributed implementation of exchange matching.